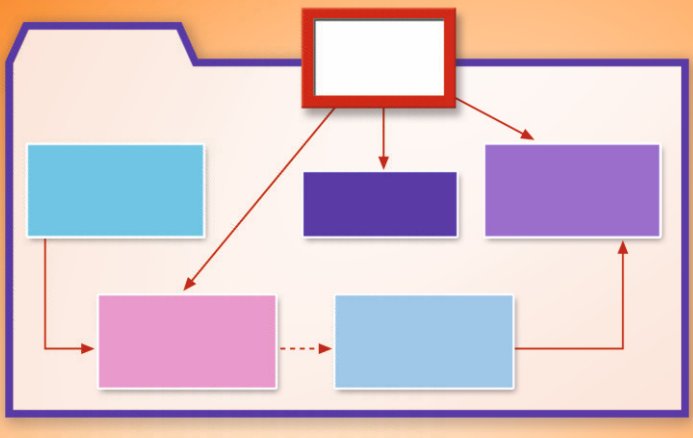
Karl Eilebrecht Gernot Starke

Patterns kompakt

Entwurfsmuster für effektive Softwareentwicklung

5. Auflage



Springer Vieweg

IT kompakt

Werke der „kompakt-Reihe“ zu wichtigen Konzepten und Technologien der IT-Branche:

- ermöglichen einen raschen Einstieg,
- bieten einen fundierten Überblick,
- sind praxisorientiert, aktuell und immer ihren Preis wert.

Weitere Bände in der Reihe
http://www.springer.com/series/8297

Karl Eilebrecht · Gernot Starke

Patterns kompakt

Entwurfsmuster für effektive Softwareentwicklung

5., aktualisierte und erweiterte Auflage

Springer

Karl Eilebrecht
Karlsruhe, Deutschland

Gernot Starke
Köln, Deutschland

ISSN 2195-3651
IT kompakt
ISBN 978-3-662-57936-7
https://doi.org/10.1007/978-3-662-57937-4

ISSN 2195-366X (electronic)
ISBN 978-3-662-57937-4 (eBook)

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

SPVI

Springer Vieweg ist ein Imprint der eingetragenen Gesellschaft Springer-Verlag GmbH, DE und ist ein Teil von Springer Nature.
Die Anschrift der Gesellschaft ist: Heidelberger Platz 3, 14197 Berlin, Germany

Einleitung

This book is meant to be played,
rather than to be read in an armchair.
Jerry Coker et al.:
Patterns for Jazz, Studio P/R, 1970

Wozu benötigen wir Entwurfsmuster?

Entwurfsmuster lösen bekannte, wiederkehrende Entwurfsprobleme. Sie fassen Design- und Architekturwissen in kompakter und wiederverwertbarer Form zusammen. Sowohl Softwareentwicklern als auch Softwarearchitekten bieten Entwurfsmuster wertvolle Unterstützung bei der Wiederverwendung erprobter Designentscheidungen. Sie geben Hinweise, wie Sie vorhandene Entwürfe flexibler, verständlicher oder auch performanter machen können.

In komplexen Software-Projekten kann der angemessene Einsatz von Mustern das Risiko von Entwurfsfehlern deutlich senken.

Warum ein weiteres Buch über Entwurfsmuster?

Seit dem Kultbuch der berühmten „Gang-of-Four“ [GoF] hat es viele Konferenzen und noch mehr Literatur zu diesem Thema gegeben – der Fundus an verfügbaren Entwurfsmustern scheint nahezu grenzenlos: mehrere tausend Druckseiten, viele hundert Seiten im Internet. Für Praktiker inmitten von Projektstress und konkreten Entwurfsproblemen stellt sich das Problem, aus der Fülle der verfügbaren Muster die jeweils geeigneten auszuwählen. Softwarearchitekten, -designer und -entwickler benötigen Unterstützung bei konkreten Entwurfsproblemen, und das auf möglichst engem Raum konzentriert.

Für solche Situationen haben wir dieses Buch geschrieben: Es erleichtert den Entwurf flexibler, wartbarer und performanter Anwendungen,

indem es das Wissen der umfangreichen Pattern-Literatur auf praxisrelevante Muster für kommerzielle Software-Systeme konzentriert. Die kompakte Darstellung erleichtert den Überblick und damit die Anwendbarkeit der ausgewählten Muster.

Ganz bewusst verzichten wir bei den vorgestellten Mustern auf ausführliche Implementierungsanleitungen und Beispielcode in gedruckter Form. Anstelle dessen erhalten Sie Hinweise auf weitere Informationsquellen und finden ausführbare Beispiele auf unserer Webseite (siehe [PK]). Die erfahrenen Praktiker unter Ihnen können anhand der kompakten Darstellung die Entwurfsentscheidung für oder gegen den Einsatz bestimmter Muster treffen. Grundlegende Kenntnisse einer objektorientierten Programmiersprache und UML setzen wir in diesem Buch voraus.

Ein Wort zur Vorsicht

Used in the wrong place,
the best patterns will fail.
Jerry Coker et al.:
Patterns for Jazz, Studio P/R, 1970

Patterns eignen sich hervorragend zur Kommunikation über Entwurfsentscheidungen. Sie können helfen, Ihre Entwürfe flexibler zu gestalten. Häufig entstehen durch die Anwendung von Patterns jedoch zusätzliche Klassen oder Interfaces, die das System aufblähen. Eine der wichtigsten Regeln beim Software-Entwurf lautet: Halten Sie Ihre Entwürfe so einfach wie möglich. In diesem Sinne möchten wir Sie, trotz aller Begeisterung für Entwurfsmuster, zu vorsichtigem Umgang mit diesen Instrumenten auffordern. Ein einfach gehaltener Entwurf ist leichter verständlich und übersichtlicher. Hinterfragen Sie bei der Anwendung von Mustern, ob Ihnen die Flexibilität, Performance oder Wiederverwendbarkeit nach der Anwendung eines Musters einen angemessenen Mehrwert gegenüber dem ursprünglichen Entwurf bieten. In Zweifelsfällen wählen Sie den einfacheren Weg.

Die meisten Entwurfsmuster in diesem Buch konzentrieren sich auf objektorientierte Systeme. Die Verfechter funktionaler Sprachen (etwa Haskell, Clojure), deklarativer Programmierung (etwa Prolog, SQL) oder

solcher Sprachen, die mehreren Paradigmen folgen (etwa Scala, Erlang, Go) mögen uns verzeihen, dass wir auf ihre „Lieblinge“ nicht gesondert eingehen – sonst hätten wir das Buch nicht so kompakt halten können.

Die Pattern-Schablone

Wir haben für dieses Buch bewusst eine flexible Schablone für Muster gewählt und ergänzende Informationen je nach Pattern aufgeführt.

- **Zweck**: Wozu dient das Pattern?
- **Szenario** (noch weitere Teile sind optional): Ein Beispielszenario für das Pattern oder das Problem.
- **Problem/Kontext**: Der strukturelle oder technische Kontext, in dem ein Problem auftritt und auf den die Lösung angewendet werden kann.
- **Lösung**: Die Lösung erklärt, wie das Problem im Kontext gelöst werden kann. Sie beschreibt die Struktur, hier meist durch UML-Diagramme. Christopher Alexander, Begründer der Pattern-Bewegung, selbst schreibt dazu: „Wenn Du davon kein Diagramm zeichnen kannst, dann ist es kein Muster.“ [Alexander, S. 267].
- **Vorteile**: Welche Vorteile entstehen aus der Anwendung dieses Patterns?
- **Nachteile**: In manchen Fällen können durch die Anwendung eines Musters Nachteile auftreten. Dies ist häufig der Fall, wenn gegensätzliche Aspekte (wie etwa Performance und Flexibilität) von einem Muster betroffen sind.
- **Verwendung**: Hier zeigen wir Ihnen Anwendungsgebiete, in denen das Muster seine spezifischen Stärken ausspielen kann.
- **Varianten**: Manche Patterns können in Variationen oder Abwandlungen vorkommen, die wir Ihnen in diesem (optionalen) Abschnitt darstellen.
- **Verweise**: Dieser Abschnitt enthält Verweise auf verwandte Muster sowie auf weiterführende Quellen.

Sorry

Ungewöhnlich – eine Entschuldigung am Anfang eines Buches: Wir möchten Sie schon jetzt um Nachsicht bitten, falls wir Ihr Lieblings-Pattern in unserer Darstellung ignorieren. Leider müssen wir aus dem riesigen Fundus interessanter Patterns eine relativ kleine Auswahl treffen – und daher viele Muster außen vorlassen. Es tut uns wirklich leid um Blackboard (eine wichtige Grundlage regel- oder wissensbasierter Systeme), um die vielen Patterns der Systemintegration, um die technikspezifischen Muster und Idiome und viele andere. Aber in diesem Buch finden Sie eine solide Basis und Verweise auf weitere interessante Literatur – und wir bleiben natürlich dran, versprochen!

Danksagung

Wir bedanken uns bei Pattern-Erfindern, in erster Linie der Gang-of-Four, Martin Fowler und Robert „Uncle Bob“ Martin sowie den zahlreichen Autoren der {Euro|Viking|Chili|.*} PloP-Konferenzen. Ihre Kreativität und Offenheit hat die Software-Welt besser gemacht! Herzlichen Dank auch unseren zahlreichen Kollegen sowie Seminar- und Schulungsteilnehmern für die fruchtbaren Diskussionen über Softwarearchitekturen, Software-Entwurf, Patterns und grünen Tee. Stefan Wießner und Jürgen Bloß aus dem KOMPR-Team sei gedankt für Espresso und wertvolle Einsichten. Michael „Agent-M“ Krusemark sowie Wolfgang Korn leisteten Erste Hilfe in C++. Schließlich geht unser Dank an Karsten Himmer nach Berlin für den ersten Hamster auf Pattern-Basis. Dr. Martin Haag und Markus Woll riskierten freundlicherweise vorab einen prüfenden Blick auf die Neuerungen der zweiten Auflage. Dmitry Dodin gab großartige „Formelumformungsnachhilfe“. Allen Lesern, die so fleißig Verbesserungsvorschläge per E-Mail geschickt haben, an dieser Stelle ein kollektives „Danke und weiter so“!

K. E.: Ich bedanke mich bei meinen Kollegen für anregende Diskussionen und natürlich bei meinen Eltern für ihre Geduld.

G. S.: Ich danke meiner Traumfrau Cheffe Uli sowie meinen Kindern Lynn und Per, ihr seid die bestmögliche Familie. Danke auch an Karl – es macht riesigen Spaß, mit Dir Bücher zu schreiben!

Inhaltsverzeichnis

1 Grundlagen des Software-Entwurfs 1
1.1 Entwurfsprinzipien 1
1.2 Heuristiken des objektorientierten Entwurfs 9
1.3 Grundprinzipien der Dokumentation 14

2 Grundkonstrukte der Objektorientierung in Java, C# und C++ 19
2.1 Vererbung 19
2.2 Abstrakte Klassen 20
2.3 Beispiel: Ein Modell von Fahrzeugen 20

3 Erzeugungsmuster 25
3.1 Abstract Factory (Abstrakte Fabrik) 25
3.2 Builder (Erbauer) 29
3.3 Factory Method (Fabrik-Methode) 34
3.4 Singleton 38
3.5 Object Pool 43

4 Verhaltensmuster 51
4.1 Command 51
4.2 Command Processor 54
4.3 Iterator 56
4.4 Visitor (Besucher) 60
4.5 Strategy 66

4.6 Template Method (Schablonenmethode) 68
4.7 Observer 70

5 Strukturmuster 77
5.1 Adapter 77
5.2 Bridge 79
5.3 Decorator (Dekorierer) 83
5.4 Fassade 87
5.5 Proxy (Stellvertreter) 89
5.6 Model View Controller (MVC) 92
5.7 Flyweight 96
5.8 Composite (Kompositum) 102

6 Verteilung 105
6.1 Combined Method 105
6.2 Data Transfer Object (DTO, Transferobjekt) 110
6.3 Transfer Object Assembler 115
6.4 Active Object 118
6.5 Master-Slave 122

7 Integration 127
7.1 Wrapper 127
7.2 Gateway 130
7.3 PlugIn 132
7.4 Mapper 136
7.5 Dependency Injection 138

8 Persistenz 143
8.1 O/R-Mapping 143
8.2 Identity Map 152
8.3 Lazy Load (Verzögertes Laden) 156
8.4 Coarse-Grained Lock (Grobkörnige Sperre) 159
8.5 Optimistic Offline Lock (Optimistisches Sperren) . . . 161
8.6 Pessimistic Offline Lock (Pessimistisches Sperren) . . . 166

9 Datenbankschlüssel 171
9.1 Identity Field (Schlüsselklasse) 174
9.2 Sequenzblock 177

9.3 UUID (Universally Unique IDentifier, Global eindeutiger Schlüssel) . . . 180
9.4 Hashwertschlüssel (Mostly Unique Hashed Attributes Identifier) . . . 182

10 Sonstige Patterns . . . 187
10.1 Money (Währung) . . . 187
10.2 Null-Objekt . . . 190
10.3 Registry . . . 193
10.4 Service Stub . . . 195
10.5 Value Object (Wertobjekt) . . . 197
10.6 Schablonendokumentation . . . 199
10.7 Inbetriebnahme . . . 204

11 Patterns – Wie geht es weiter? . . . 221
11.1 Patterns erleichtern Wissenstransfer . . . 221

Literatur . . . 229

Sachverzeichnis . . . 233

Kolophon . . . 239

Grundlagen des Software-Entwurfs 1

Für den Entwurf von Softwaresystemen gelten einige fundamentale Prinzipien, die auch die Basis der meisten Entwurfsmuster bilden. Im Folgenden stellen wir Ihnen einige dieser Prinzipien kurz vor. Detaillierte Beschreibungen finden Sie in [Riel, Eckel, Fowler, Martin2].

Als weitere Hilfe stellen wir Ihnen einige Heuristiken vor: allgemeine Leitlinien, die Sie in vielen Entwurfssituationen anwenden können. [Rechtin] und [Riel] bieten umfangreiche Sammlungen solcher Heuristiken zum Nachschlagen. [Rechtin] bezieht sich dabei grundsätzlich auf (System-)Architekturen, [Riel] nur auf objektorientierte Systeme.

Sowohl Prinzipien als auch Heuristiken können Ihnen helfen, sich in konkreten Entwurfssituationen für oder gegen die Anwendung eines Entwurfsmusters zu entscheiden.

1.1 Entwurfsprinzipien

Einfachheit vor Allgemeinverwendbarkeit

Bevorzugen Sie einfache Lösungen gegenüber allgemeinverwendbaren. Letztere sind in der Regel komplizierter. Machen Sie normale Dinge einfach und besondere Dinge möglich.

K. Eilebrecht, G. Starke, *Patterns kompakt*, IT kompakt,
https://doi.org/10.1007/978-3-662-57937-4_1

Prinzip der minimalen Verwunderung

(Principle of least astonishment) Erstaunliche Lösungen sind meist schwer verständlich.

Vermeiden Sie Wiederholung

(DRY: Don't Repeat Yourself, OAOO: Once And Once Only) Vermeiden Sie Wiederholungen von Struktur und Logik, wo sie nicht unbedingt notwendig sind.

Prinzip der einzelnen Verantwortlichkeit

(Single-Responsibility Principle, Separation-of-Concerns) Jeder Baustein eines Systems sollte eine klar abgegrenzte Verantwortlichkeit besitzen. Auf objektorientierte Systeme gemünzt heißt das: Vermeiden Sie es, Klassen mehr als eine Aufgabe zu geben. Robert Martin formuliert es so: „Jede Klasse sollte nur genau einen Grund zur Änderung haben." [Martin, S. 95]. Beispiele für solche Verantwortlichkeiten (nach [Larman]):

- Etwas wissen; Daten oder Informationen über ein Konzept kennen.
- Etwas können; Steuerungs- oder Kontrollverantwortung.
- Etwas erzeugen.

Das Prinzip ist auch auf Methodenebene anwendbar. Eine Methode sollte für eine bestimmte Aufgabe zuständig sein und nicht (durch Parameter gesteuert) für mehrere. [Martin2] legt das sehr streng aus und fordert sogar den Verzicht auf boolesche Steuer-Flags. Eine Methode wie `writeOutput(Data data, boolean append)` müsste dann in zwei Methoden gesplittet werden. Wir sehen das etwas weniger streng und empfehlen, dass eine Methode eine durch Methodennamen und Kommentar ersichtliche Aufgabe erfüllen und keine undokumentierten Seiteneffekte haben sollte. Unter der Überschrift *Command/Query-Separation Principle* fordert [Meyer], dass eine Methode, die eine

Information über ein Objekt liefert, nicht gleichzeitig dessen Zustand ändern soll.

Offen-Geschlossen-Prinzip

(Open-Closed Principle) Software-Komponenten sollten offen für Erweiterungen, aber geschlossen für Änderungen sein. Es ist eleganter und robuster, einen Klassenverbund durch Hinzufügen einer Klasse zu erweitern, als den bestehenden Quellcode zu modifizieren. Dieses Prinzip ist eng verwandt mit dem Prinzip der einzelnen Verantwortlichkeit und ebenso auf Methodenebene anwendbar. Eine entsprechende Klasse oder Methode wird *nur* im Rahmen der Verbesserung oder Korrektur der Implementierung geändert (geschlossen für Änderungen). Neue Funktionalität wird dagegen durch eine neue Klasse bzw. Methode hinzugefügt (offen für Erweiterung). Einige Patterns (z. B. ➔ Strategy (Abschn. 4.5) oder ➔ PlugIn (Abschn. 7.3)) unterstützen dieses Prinzip.

Prinzip der gemeinsamen Wiederverwendung

(Common Reuse Principle) Die Klassen innerhalb eines Pakets sollten gemeinsam wiederverwendet werden. Falls Sie eine Klasse eines solchen Pakets wiederverwenden, können Sie alle Klassen dieses Pakets wiederverwenden. Anders formuliert: Klassen, die gemeinsam verwendet werden, sollten in ein gemeinsames Paket verpackt werden. Dies hilft, zirkuläre Abhängigkeiten zwischen Paketen zu vermeiden.

Keine zirkulären Abhängigkeiten

(Acyclic Dependency Principle) Klassen und Pakete sollten keine zirkulären (zyklischen) Abhängigkeiten enthalten (siehe Abb. 1.1). Solche Zyklen sollten unter Software-architekten Teufelskreise heißen: Sie erschweren die Wartbarkeit und verringern die Flexibilität, unter anderem, weil Zyklen nur als Ganzes testbar sind. In objektorientierten Systemen können Sie zirkuläre Abhängigkeiten entweder durch Verschieben

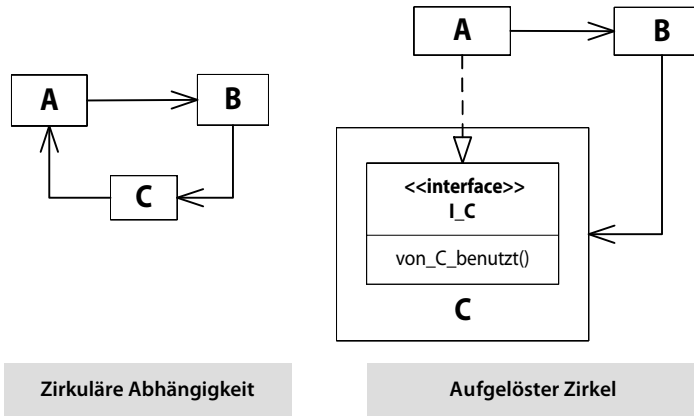


Abb. 1.1 Auflösung von Abhängigkeitszyklen

einzelner Klassen oder Methoden auflösen, oder Sie kehren eine der Abhängigkeiten durch eine Vererbungsbeziehung um.

Prinzip der stabilen Abhängigkeiten

(Stable Dependencies Principle) Führen Sie Abhängigkeiten möglichst in Richtung stabiler Bestandteile ein. Vermeiden Sie Abhängigkeiten von volatilen (d. h. häufig geänderten) Bestandteilen.

Liskov'sches Substitutionsprinzip

(Liskov Substitution Principle) Unterklassen sollen anstelle ihrer Oberklassen einsetzbar sein. Sie sollten beispielsweise in Unterklassen niemals Methoden der Oberklassen durch leere Implementierungen überschreiben. Stellen Sie beim Überschreiben von Methoden aus einer Oberklasse sicher, dass die Unterklasse in jedem Fall für die Oberklasse einsetzbar bleibt. Denken Sie besonders beim Design von Basisklassen und

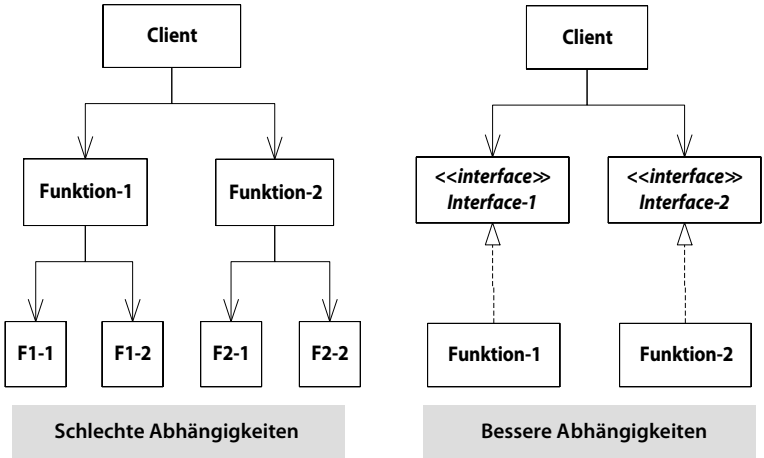


Abb. 1.2 Umkehrung von Abhängigkeiten

Interfaces an dieses Prinzip. Unbedacht eingeführte Methoden, die später doch nicht für alle Mitglieder der Klassenhierarchie passen, werden Sie nur schwer wieder los.

Prinzip der Umkehrung von Abhängigkeiten

(Dependency Inversion Principle) Nutzer einer Dienstleistung sollten möglichst von Abstraktionen (d. h. abstrakten Klassen oder Interfaces), nicht aber von konkreten Implementierungen abhängig sein (siehe Abb. 1.2). Abstraktionen sollten nicht von konkreten Implementierungen abhängen.

Prinzip der Abtrennung von Schnittstellen

(Interface Segregation Principle) Clients sollten nicht von Diensten abhängen, die sie nicht benötigen (siehe Abb. 1.3). Interfaces gehören ihren

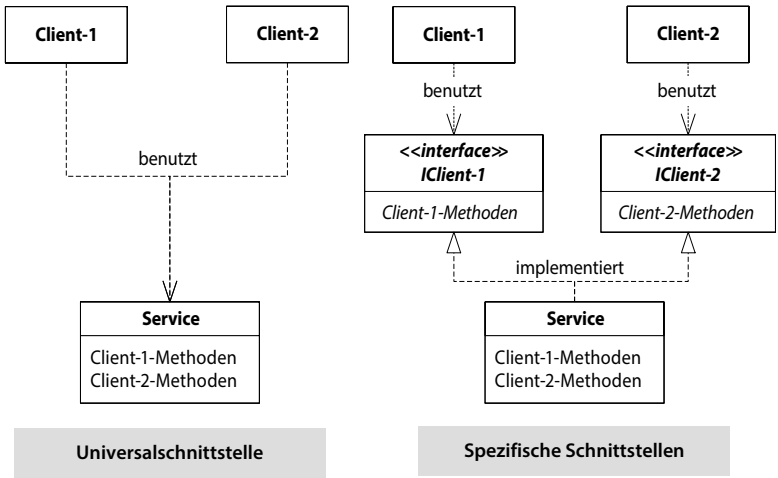


Abb. 1.3 Abtrennung von Schnittstellen

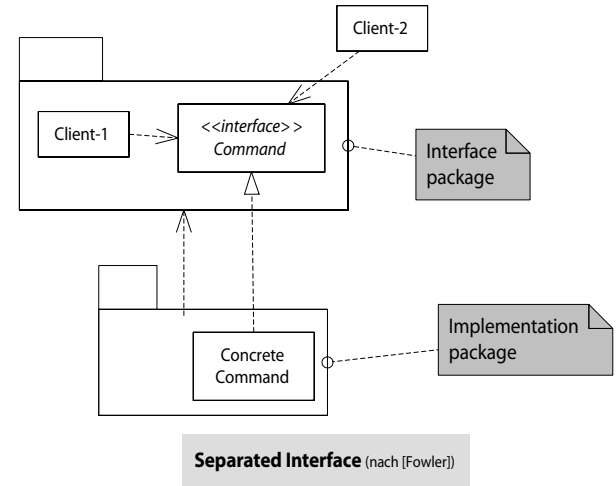


Abb. 1.4 Paket-Trennung: Separate Implementierung

Clients, nicht den Klassenhierarchien, die diese Interfaces implementieren. Entwerfen Sie Schnittstellen nach den Clients, die diese Schnittstellen brauchen.

In [Fowler] finden Sie eine Variante dieses Prinzips unter der Bezeichnung *Separated Interface* als Pattern beschrieben (siehe Abb. 1.4). Die Schnittstellen liegen dabei von ihren Implementierungen getrennt in eigenen Paketen. Zur Laufzeit müssen Sie eine konkrete Implementierung auswählen. Dafür bieten sich ➔ Erzeugungsmuster (Kap. 3) oder auch ➔ PlugIn (Abschn. 7.3) an.

Prinzip solider Annahmen

Bauen Sie Ihr Haus nicht auf Sand! Die Gefahr versteckter Annahmen zieht sich durch den gesamten Prozess der Software-Entwicklung. Das beginnt bereits damit, dass Sie nicht stillschweigend davon ausgehen können, dass Sie vorhandene Strukturen wie einen LDAP für Ihre Applikation mitbenutzen dürfen. Besonders unangenehm können Annahmen auch auf der Implementierungsebene sein. Vor einiger Zeit wurde in einem Forum eine Strategie beschrieben, mit der angeblich ein bekanntes Problem mit der Java Garbage Collection umgangen werden könnte. Ein wenig skeptisch, aber doch neugierig, bat ich (K. E.) um eine Erklärung anhand von Fakten wie der Spezifikation oder weiterführender Dokumentation. Das Resultat war eine ziemliche Abfuhr: „[...] Great. Feel free to program to what the API says. And I will continue to program to what the API actually does [...]“. Nun, auf den ersten Blick klingt diese Argumentation geradezu verlockend plausibel. Andererseits machen Sie sich dabei jedoch abhängig von einer ganz bestimmten Implementierung – ohne jede Garantie. Interfaces verlieren ihren Sinn, es bildet sich ein Nährboden für Legenden und versteckte Fehlerquellen, bevorzugt in Verbindung mit verschiedenen Releases oder Plattformen.

Nicht fundierte Annahmen über Systeme oder Schnittstellen sollten Sie also unbedingt vermeiden!

Falls dies nicht möglich ist, z. B. bei der Verwendung unfertiger oder mangelhaft dokumentierter Module bzw. bei der Erstellung von Prototypen, dokumentieren Sie Ihre Annahmen sorgfältig.

Konvention vor Konfiguration

(Convention over Configuration, Configuration by Exception) Bei diesem Ansatz wird dem Verwender eines Frameworks oder einer Software-Komponente die Konfiguration durch Konventionen und sinnvolle Voreinstellungen erleichtert.

In der Software-Entwicklung sind, bedingt durch den Einsatz komplexer Frameworks, sehr viele Einstellungen konfigurierbar, häufig durch XML-Dateien. Der Höhepunkt war wohl mit EJB 2.1 erreicht. Entwickler waren schon fast genötigt, weitere Frameworks (z. B. xdoclet) einzusetzen, um die Konfigurationsseuche halbwegs in den Griff zu bekommen. Bedingt durch die Einführung von Annotationen im Quellcode und mit EJB 3 ist vieles besser geworden, die Ursprungsfrage bleibt aber: Warum muss ich etwas aktiv konfigurieren, wenn es doch in 99 % aller Fälle immer gleich ist, einem einfachen Schema folgt oder schlimmstenfalls für meine Anwendung völlig irrelevant ist?

Moderne Frameworks wie Spring (http://www.spring.io/) und insbesondere Spring Boot drehen den Spieß um. Sie führen für Framework-Einstellungen oder auch das Mapping von Entitäten (siehe auch ➔ O/R-Mapping (Abschn. 8.1)) strikte Konventionen und Standardwerte ein. Hält sich der Framework-Verwender an die Konventionen, muss er nur noch vergleichsweise wenig individuell einstellen. Nur in den Fällen, in denen das geplante Szenario ein Abweichen von den Konventionen erfordert, ist Handarbeit nötig – dann allerdings meist viel. Dieses Paradigma ist nicht nur auf Frameworks sondern auch auf kleinere Komponenten anwendbar, sofern diese Konfigurationseinstellungen vorsehen. Machen Sie sich Gedanken über Konventionen und sinnvolle Defaultwerte. Das erleichtert anderen, Ihre Software zu evaluieren und zu integrieren. Nachteilig ist, dass viel Arbeit in der Ausarbeitung langlebiger Konventionen steckt. Zudem werden Weiterentwicklungen und Änderungen aufwendiger.

Einen schönen Artikel dazu finden Sie unter http://softwareengineering.vazexqi.com/files/pattern.html.

1.2 Heuristiken des objektorientierten Entwurfs

Entwurf von Klassen und Objekten

- Eine Klasse sollte genau eine Abstraktion („Verantwortlichkeit“) realisieren.
- Kapseln Sie zusammengehörige Daten und deren Verhalten in einer gemeinsamen Klasse (*Maximum Cohesion*).
- Wenn Sie etwas Schlechtes, Schmutziges oder Unschönes tun müssen, dann kapseln Sie es zumindest in einer einzigen Klasse.
- Kapseln Sie Aspekte, die variieren können. Falls Ihnen dies zu abstrakt ist: Das ➔ Bridge-Pattern (Abschn. 5.2) wendet diese Heuristik an.
- Vermeiden Sie übermäßig mächtige Klassen („Poltergeister“ oder „Gott-Klassen“).
- Benutzer einer Klasse dürfen ausschließlich von deren öffentlichen Schnittstellen abhängen. Klassen sollten nicht von ihren Benutzern abhängig sein.
- Halten Sie die öffentlichen Schnittstellen von Klassen möglichst schlank. Entwerfen Sie so privat wie möglich.
- Benutzen Sie Attribute, um Veränderungen von Werten auszudrücken. Um Veränderung im Verhalten auszudrücken, können Sie Überlagerung von Methoden verwenden.
- Vermeiden Sie es, den Typ eines Objekts zur Laufzeit zu ändern. Einige Sprachen erlauben dies zwar mittels mehr oder weniger gut dokumentierter Hacks. Dies geschieht dann aber in der Absicht, eine existierende Instanz zu einem anderen Typ kompatibel zu machen. Es ist deutlich besseres Design, ein solches Problem mit dem ➔ Decorator-Pattern (Abschn. 5.3) oder dem State-Pattern (s. [GoF]) zu lösen. Das .NET-Framework bietet zudem die Möglichkeit der *Extension-Methods* [Troelson]. In einigen Sprachen können Sie (nachträglich) eine implizite Konvertierung definieren (z. B. *Scala implicits*), was zu kurzem, sehr elegantem Code führt. Wem aber schon einmal auf mysteriöse Weise Kommastellen in einer SQL-Berechnung abhanden gekommen sind, der weiß, dass alle implizite weiße Magie ihre Schattenseiten hat!

- Hüten Sie sich davor, Objekte einer Klasse als Unterklassen zu modellieren. In der Regel sollte es von abgeleiteten Klassen mehr als eine einzige Instanz geben können. Ausgenommen sind *Algebraische Datentypen* (siehe [EKL2011]) wie z. B. Enumerationen.
- Wenn Klassen besonders viele `getX()`-Methoden in der öffentlichen Schnittstelle enthalten, haben Sie möglicherweise die Zusammengehörigkeit von Daten und Verhalten verletzt.
- Halten Sie sich bei der Modellierung von Klassen möglichst nah an die reale Welt. Streben Sie bei Analyse- und Design-Modellen nach strukturellen Ähnlichkeiten.
- Vermeiden Sie überlange Argumentlisten. Darunter leidet die Übersichtlichkeit von Methodenaufrufen. Verschieben Sie die Methode in eine andere Klasse oder übergeben Sie höher aggregierte Objekte als Argumente.

Beziehungen zwischen Klassen und Objekten

- Minimieren Sie die Abhängigkeiten einer Klasse (*Minimal Coupling*).
- Viele Methoden einer Klasse sollten viele Attribute dieser Klasse häufig benutzen. Anders formuliert: Wenn viele Methoden nur mit wenigen Attributen arbeiten, dann haben Sie möglicherweise die Zusammengehörigkeit von Daten und Verhalten verletzt.
- Eine Klasse sollte nicht wissen, worin sie enthalten ist.
- Regel von Demeter (Law of Demeter, LoD): „Reden Sie nicht mit Fremden“, d. h., ein Objekt sollte nur sich selbst, seine Attribute oder die Argumente seiner Methoden referenzieren. Kennen Sie das Spiel Halma? Man springt möglichst weite Pfade über viele Spielsteine hinweg, um schnell ans Ziel zu kommen. Diese *transitive Navigation* sollten Sie bei der Software-Entwicklung tunlichst vermeiden [Martin2]. Aufrufe wie `outputHandler.getCurrentDestination().getFile().getSize()` zeigen einen Verstoß gegen LoD und deuten auf ein schlechtes oder unvollständiges Interface hin. Im Beispiel könnten Sie das Design verbessern, indem Sie die Klasse `OutputHandler` um eine Methode `getBytesWritten()` erweitern.

Vererbung und Delegation

- Verwenden Sie Vererbung nur zur Modellierung von Spezialisierungen.
- Keine Oberklasse sollte etwas über ihre Unterklassen wissen.
- Abstrakte Klassen sollten Basisklassen ihrer Hierarchie sein bzw. nur von abstrakten Klassen ableiten.
- Falls mehrere Klassen A und B nur gemeinsame Daten besitzen (aber kein gemeinsames Verhalten), dann sollten diese gemeinsamen Daten in einer eigenen Klasse sein, die in A und B enthalten ist.
- Vermeiden Sie es, den Typ einer Klasse explizit zu untersuchen. Verwenden Sie stattdessen Polymorphismus.
- Überschreiben Sie niemals eine Methode einer Oberklasse mit einer leeren Implementierung (das führt zu einer Verletzung des Liskov'schen Substitutionsprinzips).
- Vermeiden Sie Mehrfachvererbung. Genauer: Vermeiden Sie mehrfache Implementierungsvererbung. Mehrfache Schnittstellenvererbung hingegen ist unproblematisch.
- Bevorzugen Sie, wenn möglich, Schnittstellen (*Interfaces*) gegenüber abstrakten Klassen.

Verteilung

- Martin Fowlers Heuristik zur verteilten Datenverarbeitung: Vermeiden Sie Verteilung, wo immer möglich. Die Welt ist auch ohne Verteilung komplex genug (vgl. [Fowler]).
- Angemessen eingesetzt kann Verteilung die Flexibilität oder Skalierbarkeit von Systemen verbessern.

Nebenläufigkeit

(Concurrency) Moderne Mehrkernprozessoren lassen sich mit einem einzelnen Thread nicht auslasten. Schon deshalb lohnt sich die Beschäftigung mit dem Thema. Leider ist und bleibt Multithreading kompliziert.

Es ist ein bisschen wie mit Zeitreisen: Solange Sie nichts anfassen und mit niemandem reden, kann eigentlich nichts schief gehen. Andernfalls *können* interessante Dinge passieren (fragen Sie mal Homer Simpson, in „Zeit und Sühne"). Sie sollten also gut überlegen, ob Sie im geplanten Szenario wirklich Nebenläufigkeit benötigen.

- Schätzen Sie ab, ob Sie durch mehrere Threads wirklich etwas gewinnen. Ist der Zeitgewinn relevant?
- Entwickeln Sie zunächst eine funktionierende Lösung mit nur einem Thread, erst dann eine mit mehreren.
- Müssen die Threads über *shared state* kommunizieren oder bietet sich Messaging (*actor based concurrency*) an? Letzteres erfordert zwar ein Umdenken in der Entwicklung, schirmt Sie aber vor vielen Seiteneffekten des Zugriffs auf gemeinsam genutzte Speicherbereiche ab. Eine Reihe von Sprachen und Frameworks unterstützen dieses Konzept, Beispiele sind Scala, F#, GParallelizer (für Groovy) oder Kilim (für Java). Allen Interessierten empfehlen wir zudem, sich mit dem Aktor-Modell (etwa: Akka-Framework) oder den Goroutines der Sprache Golang zu beschäftigen. Dort sehen Sie, wie Nebenläufigkeit zumindest ein ganzes Stück einfacher und risikoärmer funktionieren kann, als das beispielsweise in Java der Fall ist.
- Prüfen Sie, ob es hinsichtlich Speicherverbrauch, Performance und Konsistenz akzeptabel ist, die Threads mit exklusiven Kopien der Daten arbeiten zu lassen (vgl. [Martin2]). Damit können Sie die Problematik des *shared state* umgehen oder zumindest die Zahl der Synchronisationspunkte reduzieren.
- Sie sollten bei der Einführung jedes neuen Objekts abwägen, ob Sie seine Attribute final deklarieren können. Solche unveränderlichen (immutable) Instanzen sind *consistent by design*. Das klingt banal, aber ich (K. E.) sehe immer wieder, wie dieser Vorteil leichtfertig verschenkt wurde. Späteres Refactoring in diese Richtung ist meist mühsam.
- Informieren Sie sich, was die von ihnen präferierte Sprache (z. B. Java, C# etc.) oder Umgebung (z. B. Applikationsserver) zu bieten hat. Teilprobleme wurden wahrscheinlich bereits von klugen Köpfen